# Non-interactive Proofs of Proof-of-Work

Aggelos Kiayias[1], Andrew Miller[2], and Dionysis Zindros[3](✉)

[1] University of Edinburgh, IOHK, Edinburgh, Scotland
akiayias@inf.ed.ac.uk
[2] University of Illinois at Urbana-Champaign,
Initiative for Cryptocurrencies and Contracts, Urbana, USA
amiller@cs.umd.edu
[3] National and Kapodistrian University of Athens, IOHK, Athens, Greece
dionyziz@di.uoa.gr

**Abstract.** Decentralized consensus protocols based on proof-of-work (PoW) mining require nodes to download data linear in the size of the blockchain even if they make use of Simplified Payment Verification (SPV). In this work, we put forth a new formalization of proof-of-work verification by introducing a primitive called Non-Interactive Proofs of Proof-of-Work (NIPoPoWs). We improve upon the previously known SPV NIPoPoW by proposing a novel NIPoPoW construction using superblocks, blocks that are much heavier than usual blocks, which capture the fact that proof-of-work took place without sending all of it. Unlike a traditional blockchain client which must verify the entire linearly-growing chain of PoWs, clients based on superblock NIPoPoWs require resources only logarithmic in the length of the chain, instead downloading a compressed form of the chain. Superblock NIPoPoWs are thus *succinct* proofs and, due to their non-interactivity, require only a single message between the prover and the verifier of the transaction. Our construction allows the creation of *superlight* clients which can synchronize with the network quickly even if they remain offline for large periods of time. Our scheme is provably secure in the Bitcoin Backbone model. From a theoretical point of view, we are the first to propose a cryptographic prover–verifier definition for decentralized consensus protocols and the first to give a construction which can synchronize non-interactively using only a logarithmically-sized message.

## 1 Introduction

Proof-of-work blockchain clients such as mobile wallets today are based on the Simplified Payment Verifications (SPV) protocol, which was described in the original Bitcoin paper [14], and allows them to synchronize with the network by downloading only block headers and not the entire blockchain with transactions. However, such initial synchronization still requires receiving all the block headers. In this work, we study the question of whether better protocols exist and in particular if downloading fewer block headers is sufficient to securely synchronize with the rest of the blockchain network. Our requirement is that the system remains decentralized and that useful facts about the blockchain (such as the Merkle root of current account balances in Ethereum [5,19]) can be deduced from the downloaded data.

**Our Contributions.** We put forth a cryptographic security definition for Non-Interactive Proofs of Proof-of-Work protocols which describes what such a synchronization protocol must achieve (Sect. 2). We then construct a protocol which solves the problem and requires sending only a logarithmic number of blocks from the chain. We construct a protocol which can synchronize recent blocks, the *suffix proofs* protocol (Sect. 4). We analyze the security and succinctness of our protocol in Sect. 5. In Sect. 6, we show a simple addition to the suffix proofs protocol which allows synchronizing any part of the blockchain that the client may be interested in, the *infix proofs* protocol.

**Previous Work.** The need for succinct clients was first identified by Nakamoto in his original paper [14]. Predicates pertaining to events occurring in the blockchain have been explored in the setting of sidechains [2]. It has also been implemented for simple classes of predicates such as atomic swaps [10,15], which do not allow full synchronization. Non-succinct certificates about proof-of-*stake* blockchains have been proposed in [8], but their scheme is not applicable to proof-of-*work*. Superblocks were first described in the Bitcoin Forum [13] and later formalized [11] to describe their *Proofs of Proof-of-Work* which have limited applications due to interactivity, lack of security, and inability to prove facts buried deep within the blockchain. We improve upon their work with a security definition, an interactive construction, and an attack against their scheme which works with overwhelming probability.

## 2 Model and Definitions

Our model is based on the "backbone" model for proof-of-work cryptocurrencies [7], extended with SPV. Following their model, we assume *synchrony* (*partial synchrony* with *bounded delay* [16] is left for future work) and constant difficulty.

**Backbone Model.** The entities on the blockchain network are of 3 kinds: (1) Miners, who try to mine new blocks on top of the longest known blockchain and broadcast them as soon as they are discovered. Miners commit new transactions they receive from clients. (2) Full nodes, who maintain the longest blockchain without mining and also act as the provers in the network. (3) Verifiers or stateless clients, who do not store the entire blockchain, but instead connect to provers and ask for proofs in regards to which blockchain is the largest. The verifiers attempt to determine the value of a predicate on these chains, for example whether a particular payment has been finalized.

Our main challenge is to design a protocol so that clients can sieve through the responses they receive from the network and reach a conclusion that should never disagree with the conclusion of a full node who is faced with the same objective and infers it from its local blockchain state.

We model proof-of-work discovery attempts by using a random oracle [3]. The random oracle produces $\kappa$-bit strings, where $\kappa$ is the system's security parameter. The network is synchronized into numbered rounds, which correspond to moments in time. $n$ denotes the total number of miners in the game, while $t$ denotes the total number of adversarial miners. Each miner is assumed to have equal mining power captured by the number of queries $q$ available per player to the random oracle per round, each query of which succeeds independently with probability $p$ (a successful query produces a block with valid proof-of-work). Mining pools and miners of different computing power can be captured by assuming multiple players combine their computing power. This is made explicit for the adversary, as they do not incur any network overhead to achieve communication between adversarial miners. On the contrary, honest players discovering a block must *diffuse* it (broadcast it) to the network at a given round and wait for it to be received by the rest of the honest players at the beginning of the next round. A round during which an honest block is diffused is called a *successful round*; if the number of honest blocks diffused is one, it is called a *uniquely successful round*. We assume there is an honest majority, i.e., that $t/n < 0.5$ with a constant minimum gap [7]. We further assume the network is adversarial, but there is no eclipsing attacks [9]. More specifically, we allow the adversary to reorder messages transmitted at a particular round, to inject new messages thereby capturing Sybil attacks [6], but not to drop messages. Each honest miner maintains a local chain $\mathcal{C}$ which they consider the current active blockchain. Upon receiving a different blockchain from the network, the current active blockchain is changed if the received blockchain is longer than the currently adopted one. Receiving a different blockchain of the same length as the currently adopted one does not change the adopted blockchain.

Blockchain blocks are generated by including the following data in them: *ctr*, the nonce used to achieve the proof-of-work; $x$ the Merkle tree [12] root of the transactions confirmed in this block; and *interlink* [11], a vector containing pointers to previous blocks, including the id of the previous block. The *interlink* data structure contains pointers to more blocks than just the previous block. We will explain this further in Sect. 3. Given two hash functions $H$ and $G$ modelled as random oracles, the id of a block is defined as $\mathsf{id} = H(ctr, G(x, \mathsf{interlink}))$. In bitcoin's case, both $H$ and $G$ would be SHA256.

**The Prover and Verifier Model.** In our protocol, the nodes include a *proof* along with their responses to clients. We need to assume that clients are able to connect to at least one correctly functioning node (i.e., that they cannot be eclipsed from the network [1,9]). Each client makes the same request to every node, and by verifying the proofs the client identifies the correct response. Henceforth we will call clients *verifiers* and nodes *provers*.

The prover-verifier interaction is parameterized by a predicate (e.g. "the transaction $tx$ is committed in the blockchain"). The predicates of interest in our context are predicates on the active blockchain. Some of the predicates are more suitable for succinct proofs than others. We focus our attention in *stable* predicates having the property that all honest miners share their view of them in a way that is updated in a predictable manner, with a truth-value that persists as the blockchain grows (an example of an unstable predicate is e.g., the least significant bit of the hash of last block). Following the work of [7], we wait for $k$ blocks to bury a block before we consider it *confirmed* and thereby the predicates depending on it stable. $k$ is the *common prefix* security parameter, which in Bitcoin folklore is often taken to be $k = 6$.

In our setting, for a given predicate $Q$, several provers (including adversarial ones) will generate proofs claiming potentially different truth values for $Q$ based on their claimed local longest chains. The verifier receives these proofs and accepts one of the proofs, determining the truth value of the predicate. We denote a *blockchain proof protocol* for a predicate $Q$ as a pair $(P, V)$ where $P$ is the *prover* and $V$ is the *verifier*. $P$ is a PPT algorithm that is spawned by a full node when they wish to produce a proof, accepts as input a full chain $\mathcal{C}$ and produces a proof $\pi$ as its output. $V$ is a PPT algorithm which is spawned at some round (having only Genesis), receives a pair of proofs $(\pi_A, \pi_B)$ from both an honest party and the adversary and returns its decision $d \in \{T, F\}$ before the next round and terminates. The honest miners produce proofs for $V$ using $P$, while the adversary produces proofs following some arbitrary strategy. Before we introduce the security properties for blockchain proof protocols we introduce some necessary notation for blockchains.

**Notation.** Blockchains are finite block sequences obeying the *blockchain property*: that in every block in the chain there exists a pointer to its previous block. A chain is *anchored* if its first block is *genesis*, denoted *Gen*. For chain addressing we use Python brackets $\mathcal{C}[\cdot]$ as in [17]. A zero-based positive number in a bracket indicates the indexed block in the chain. A negative index indicates a block from the end, e.g., $\mathcal{C}[-1]$ is the tip of the blockchain. A range $\mathcal{C}[i : j]$ is a subarray starting from $i$ (inclusive) to j (exclusive). Given chains $\mathcal{C}_1, \mathcal{C}_2$ and blocks $A, Z$ we concatenate them as $\mathcal{C}_1\mathcal{C}_2$ or $\mathcal{C}_1 A$. $\mathcal{C}_2[0]$ must point to $\mathcal{C}_1[-1]$ and $A$ must point to $\mathcal{C}_1[-1]$. We denote $\mathcal{C}\{A : Z\}$ the subarray of the chain from $A$ (inclusive) to $Z$ (exclusive). We can omit blocks or indices from either side of the range to take the chain to the beginning or end respectively. The *id* function returns the id of a block given its data, i.e., $\mathsf{id} = H(ctr, G(x, \mathsf{interlink}))$.

## 2.1 Provable Chain Predicates

Our aim is to prove statements about the blockchain, such as "The transaction *tx* is included in the current blockchain" without transmitting all block headers. We consider a general class of predicates that take on values *true* or *false*. Since a Bitcoin-like blockchain can experience delays and intermittent forks, not all honest parties will be in exact agreement about the entire chain. However, when all honest parties are in agreement about the truth value of the predicate, we require that the verifier also arrives at the same truth value.

To aid the construction of our proofs, we focus on predicates that are *monotonic*; they start with the value *false* and, as the blockchain grows, can change their value to *true* but not back.

**Definition 1** *(Monotonicity).* *A chain predicate $Q(\mathcal{C})$ is* monotonic *if for all chains $\mathcal{C}$ and for all blocks $B$ we have that $Q(\mathcal{C}) \Rightarrow Q(\mathcal{C}B)$.*

Additionally, we require that our predicates only depend on the *stable* portion of the blockchain, blocks that are buried under $k$ subsequent blocks. This ensures that the value of the predicate will not change due to a blockchain reorganization.

**Definition 2** *(Stability).* *Parameterized by $k \in \mathbb{N}$, a chain predicate $Q$ is $k$-*stable *if its value only depends on the prefix $\mathcal{C}[: -k]$.*

## 2.2 Desired Properties

We now define two desired properties of a non-interactive blockchain proof protocol, *succinctness* and *security.*

**Definition 3** *(Security).* *A* blockchain proof protocol *$(P, V)$ about a predicate $Q$ is* secure *if for all environments and for all PPT adversaries $\mathcal{A}$ and for all rounds $r \geq \eta k$, if $V$ receives a set of proofs $\mathcal{P}$ at the beginning of round $r$, at least one of which has been generated by the honest prover $P$, then the output of $V$ at the end of round $r$ has the following constraints:*

- *If the output of $V$ is* false*, then the evaluation of $Q(\mathcal{C})$ for* all *honest parties must be* false *at the end of round $r - \eta k$.*
- *If the output of $V$ is* true*, then the evaluation of $Q(\mathcal{C})$ for* all *honest parties must be* true *at the end of round $r + \eta k$.*

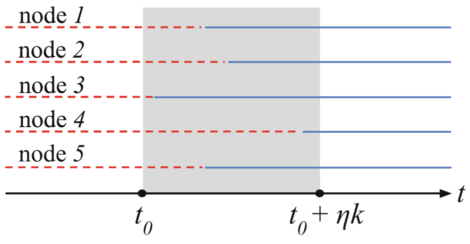

**Fig. 1.** The truth value of a fixed predicate $Q$ about the blockchain, as seen from the point of view of 5 honest nodes, drawn on the vertical axis, over time, drawn as the horizontal axis. The truth value evolves over time starting as *false* at the beginning, indicated by a dashed red line. At some point in time $t_0$, the predicate is ready to be evaluated as *true*, indicated by the solid blue line. The various honest nodes each realize this independently over a period of $\eta k$ duration, shaded in gray. The predicate remains *false* for everyone before $t_0$ and *true* for everyone after $t_0 + \eta k$. (Color figure online)

Some explanation is needed for the rationale of the above definition. The parameter $\eta$ is borrowed from the Backbone [7] work and indicates the rate at which new blocks are produced, i.e., the number of rounds needed on average to produce a block. If the scheme is secure, this means that the output of the verifier should match the output of a *potential honest full node*. However, in various executions, not all potential honest full node behaviors will be instantiated. Therefore, we require that, if the output of the proof verifier is *true* then, consistently with honest behavior, all other honest full nodes will converge to the value *true*. Conversely, if the output of the proof verifier is *false* then, consistently with honest behavior, all honest full nodes must have indicated *false* sufficiently long in the past. The period $\eta k$ is the period needed for obtaining sufficient confirmations ($k$) in a blockchain system. A predicate's value has the potential of being *true* as seen by an honest party starting at time $t_0$. Before time $t_0$, all honest parties agree that the predicate is *false*. It takes $\eta k$ time for all parties to agree that the predicate is *true*, which is certain after time $t_0 + \eta k$. The adversary may be able to convince the verifier that the predicate has any value during the period from $t_0$ to $t_0 + \eta k$. However, our security definition mandates that before time $t_0$ the verifier will necessarily output *false* and after time $t_0 + \eta k$ the verifier will necessarily output *true* (Fig. 1).

**Definition 4** *(Succinctness).* *A* blockchain proof protocol $(P, V)$ *about a predicate* $Q$ *is* succinct *if for all PPT provers* $\mathcal{A}$*, any proof* $\pi$ *produced by* $\mathcal{A}$ *at some round* $r$*, the verifier* $V$ *only reads a* $O(polylog(r))$*-sized portion of* $\pi$*.*

It is easy to construct a *secure but not succinct* protocol for any computable predicate $Q$: The prover provides the entire chain $\mathcal{C}$ as a proof and the verifier simply selects the longest chain: by the *common-prefix property* of the backbone protocol (c.f. [7]), this is consistent with the view of every honest party (as long as $Q$ depends only on a *prefix* of the chain, as we explain in more detail shortly). In fact this is how widely-used cryptocurrency clients (including SPV clients) operate today.

It is also easy to build *succinct but insecure* clients: The prover simply sends the predicate value directly. This is roughly what hosted wallets do [4].

The challenge we will solve is to provide a non-interactive protocol that at the same time achieves security and succinctness over a large class of useful predicates. We call this primitive a NIPoPoWs. Our particular instantiation for NIPoPoWs is a *superblock-based NIPoPoW construction.*

## 3 Consensus Layer Support

### 3.1 The Interlink Pointers Data Structure

In order to construct our protocol, we rely on the *interlink data structure* [11]. This is an additional hash-based data structure that is proposed to be included in the header of each block. The interlink data structure is a skip-list [18] that makes it efficient for a verifier to process a sparse subset of the blockchain, rather than only consecutive blocks.

Valid blocks satisfy the proof-of-work condition: $id \leq T$, where $T$ is the mining target. Throughout this work, we make the simplifying assumption that

$T$ is constant. Some blocks will achieve a lower id. If $id \leq \frac{T}{2^\mu}$ we say that the block is of level $\mu$. All blocks are level 0. Blocks with level $\mu$ are called *$\mu$-superblocks*. $\mu$-superblocks for $\mu > 0$ are also $(\mu - 1)$-superblocks. The level of a block is given as $\mu = \lfloor \log(T) - \log(\mathsf{id}(\mathsf{B})) \rfloor$ and denoted $level(B)$. By convention, for $Gen$ we set $id = 0$ and $\mu = \infty$.

Observe that in a blockchain protocol execution it is expected 1/2 of the blocks will be of level 1; 1/4 of the blocks will be of level 2; 1/8 will be of level 3; and $1/2^\mu$ blocks will be of level $\mu$. In expectation, the number of superblock levels of a chain $\mathcal{C}$ will be $\Theta(\log(\mathcal{C}))$ [11]. Figure 2 illustrates the blockchain superblocks starting from level 0 and going up to level 3 in case these blocks are distributed exactly according to expectation. Here, each level contains half the blocks of the level below.

We wish to connect the blocks at each level with a *previous block* pointer pointing to the most recent block of the same level. These pointers must be included in the data of the block so that proof-of-work commits to them. As the level of a block cannot be prediced before its proof-of-work is calculated, we extend the *previous block id* structure of classical blockchains to be a vector, the *interlink vector*. The interlink vector points to the most recent preceding block of every level $\mu$. Genesis is of infinite level and hence a pointer to it is included in every block. The number of pointers that need to be included per block is in expectation $\log(|\mathcal{C}|)$.

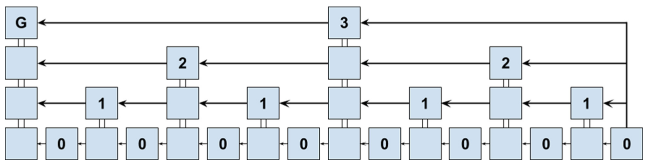

**Fig. 2.** The hierarchical blockchain. Higher levels have achieved a lower target (higher difficulty) during mining. All blocks are connected to the genesis block $G$.

The algorithm for this construction is shown in Algorithm 1 and is borrowed from [11]. The interlink data structure turns the blockchain into a skiplist-like [18] data structure.

The updateInterlink algorithm accepts a block $B'$, which already has an interlink data structure defined on it. The function evaluates the interlink data structure which needs to be included as part of the next block. It copies the existing interlink data structure and then modifies its entries from level 0 to $\mathsf{level}(B')$ to point to the block $B'$.

**Algorithm 1.** updateInterlink

```
1: function updateInterlink($B'$)
2:     interlink $\leftarrow$ $B'$.interlink
3:     for $\mu = 0$ to $level(B')$ do
4:         interlink[$\mu$] $\leftarrow$ id($B'$)
5:     end for
6:     return interlink
7: end function
```

**Traversing the Blockchain.** As we have now extended blocks to contain multiple pointers to previous blocks, if certain blocks are omitted from the middle of a chain we will obtain a subchain, as long as the *blockchain property* is maintained (i.e., that each block must contain an interlink pointer to its previous block in the sequence).

Blockchains are sequences, but it is more convenient to use set notation for some operations. Specifically, $B \in \mathcal{C}$ and $\emptyset$ have the obvious meaning. $\mathcal{C}_1 \subseteq \mathcal{C}_2$ means that all blocks in $\mathcal{C}_1$ exist in $\mathcal{C}_2$, perhaps with additional blocks intertwined. $\mathcal{C}_1 \cup \mathcal{C}_2$ is the chain obtained by sorting the blocks contained in both $\mathcal{C}_1$ and $\mathcal{C}_2$ into a sequence (this may be not always defined, as pointers may be missing). We will freely use set builder notation $\{B \in \mathcal{C} : p(B)\}$. $\mathcal{C}_1 \cap \mathcal{C}_2$ is the chain $\{B : B \in \mathcal{C}_1 \wedge B \in \mathcal{C}_2\}$. In all cases, the blockchain property must be maintained. The lowest common ancestor is $\mathsf{LCA}(\mathcal{C}_1, \mathcal{C}_2) = (\mathcal{C}_1 \cap \mathcal{C}_2)[-1]$. If $\mathcal{C}_1[0] = \mathcal{C}_2[0]$ and $\mathcal{C}_1[-1] = \mathcal{C}_2[-1]$, we say the chains $\mathcal{C}_1, \mathcal{C}_2$ *span* the same block range.

It will soon become clear that it is useful to construct a chain containing only the superblocks of another chain. Given $\mathcal{C}$ and level $\mu$, the *upchain* $\mathcal{C}{\uparrow^\mu}$ is defined as $\{B \in \mathcal{C} : level(B) \geq \mu\}$. A chain containing only $\mu$-superblocks is called a *$\mu$-superchain*. It is also useful, given a $\mu$-superchain $\mathcal{C}'$ to go back to the regular chain $\mathcal{C}$. Given chains $\mathcal{C}' \subseteq \mathcal{C}$, the *downchain* $\mathcal{C}'{\downarrow_\mathcal{C}}$ is defined as $\mathcal{C}\{\mathcal{C}'[0] : \mathcal{C}'[-1]\}$. $\mathcal{C}$ is the *underlying chain* of $\mathcal{C}'$. The underlying chain is often implied by context, so we will simply write $\mathcal{C}'{\downarrow}$. By the above definition, the $\mathcal{C}{\uparrow}$ operator is absolute: $(\mathcal{C}{\uparrow^\mu}){\uparrow^{\mu+i}} = \mathcal{C}{\uparrow^{\mu+i}}$. Given a set of consecutive rounds $S = \{r, r+1, \cdots, r+j\} \subseteq \mathbb{N}$, we define $\mathcal{C}^S = \{B \in \mathcal{C} : B \text{ was generated during } S\}$.

## 4 Non-interactive Blockchain *suffix* proofs

In this section, we introduce our non-interactive suffix proofs. With foresight, we caution the reader that the non-interactive construction we present in this section is *insecure*. A small patch will later allow us to modify our construction to achieve security.

We allow provers to prove general predicates $Q$ about the chain $\mathcal{C}$. Among the predicates which are stable, in this section, we will limit ourselves to *suffix sensitive* predicates. We extend the protocol to support more flexible predicates (such as transaction inclusion, as needed for our applications) which are not limited to the suffix in Sect. 6.

**Definition 5 (Suffix sensitivity).** *A chain predicate $Q$ is called $k$-*suffix sensitive *if its value can be efficiently computed given the last $k$ blocks of the chain.*

**Example.** In general our applications will require predicates that are not suffix-sensitive. However, as an example, consider the predicate "an Ethereum contract at address $C$ has been initialized with code $h$ at least $k$ blocks ago" where $h$ does not invoke the `selfdestruct` opcode. This can be implemented in a suffix-sensitive way because, in Ethereum, each block includes a Merkle Trie over all of the contract codes [5,19], which cannot be changed after initialization. This predicate is thus also monotonic and $k$-stable. Any predicate which is both *suffix-sensitive* and *$k$-stable* must solely depend on data at block $\mathcal{C}[-k]$.

## 4.1 Construction

We next present a generic form of the verifier first and the prover afterwards. The generic form of the verifier works with any practical suffix proof protocol. Therefore, we describe the generic verifier first before we talk about the specific instantiation of our protocol. The generic verifier is given access to call a protocol-specific proof comparison operator $\leq_m$ that we define. We begin the description of our protocol by first illustrating the generic verifier. Next, we describe the prover specific to our protocol. Finally, we show the instantiation of the $\leq_m$ operator, which plugs into the generic verifier to make a concrete verifier for our protocol.

**The Generic Verifier.** The $\mathsf{Verify}$ function of our NIPoPoW construction for suffix predicates is described in Algorithm 2. The verifier algorithm is parameterized by a chain predicate $Q$ and security parameters $k, m$; $k$ pertains to the amount of proof-of-work needed to bury a block so that it is believed to remain stable (e.g., $k = 6$); $m$ is a security parameter pertaining to the prefix of the proof, which connects the genesis block to the $k$-sized suffix. The verifier receives several proofs by different provers in a collection of proofs $\mathcal{P}$ at least one of which will be honest. Iterating over these proofs, it extracts the best.

Each proof is a chain. For honest provers, these are subchains of the adopted chain. Proofs consist of two parts, $\pi$ and $\chi$; $\pi\chi$ must be a valid chain; $\chi$ is the proof suffix; $\pi$ is the prefix. We require $|\chi| = k$. For honest provers, $\chi$ is the last $k$ blocks of the adopted chain, while $\pi$ consists of a selected subset of blocks from the rest of their chain preceding $\chi$. The method of choice of this subset will become clear soon.

**Algorithm 2.** The Verify algorithm for the NIPoPoW protocol

```
1: function $\mathsf{Verify}^{Q}_{m,k}(\mathcal{P})$
2:     $\tilde{\pi} \leftarrow (\text{Gen})$                              ▷ Trivial anchored blockchain
3:     for $(\pi, \chi) \in \mathcal{P}$ do                     ▷ Examine each proof $(\pi, \chi)$ in $\mathcal{P}$
4:         if $\mathsf{validChain}(\pi\chi) \wedge |\chi| = k \wedge \pi \geq_m \tilde{\pi}$ then
5:             $\tilde{\pi} \leftarrow \pi$
6:             $\tilde{\chi} \leftarrow \chi$                              ▷ Update current best
7:         end if
8:     end for
9:     return $\tilde{Q}(\tilde{\chi})$
10: end function
```

The verifier compares the proof prefixes provided to it by calling the $\geq_m$ operator. We will get to the operator's definition shortly. Proofs are checked for validity before comparison by ensuring $|\chi| = k$ and calling $\mathsf{validChain}$ which checks if $\pi\chi$ is an anchored blockchain.

At each loop iteration, the verifier compares the next candidate proof prefix $\pi$ against the currently best known proof prefix $\tilde{\pi}$ by calling $\pi \geq_m \tilde{\pi}$. If the candidate prefix is better than the currently best known proof prefix, then the currently known best prefix is updated by setting $\tilde{\pi} \leftarrow \pi$. When the best known prefix is updated, the suffix $\tilde{\chi}$ associated with the best known prefix is also updated to match the suffix $\chi$ of the candidate proof by setting $\tilde{\chi} \leftarrow \chi$. While $\tilde{\chi}$ is needed for the final predicate evaluation, it is not used as part of any comparison, as it has the same size $k$ for all proofs. The best known proof prefix is initially set to $(Gen)$, the trivial anchored chain containing only the genesis block. Any well-formed proof compares favourably against the trivial chain.

After the end of the `for` loop, the verifier will have determined the best proof $(\tilde{\pi}, \tilde{\chi})$. We will later prove that this proof will necessarily belong to an honest prover with overwhelming probability. Since the proof has been generated by an honest prover, it is associated with an underlying honestly adopted chain $\mathcal{C}$. The verifier then extracts the value of the predicate $Q$ on the underlying chain. Note that, because the full chain is not available to the verifier, the verifier here must evaluate the predicate on the suffix. Because the predicate is suffix-sensitive, it is possible to do so. As a technical detail, we denote $\tilde{Q}$ the predicate which accepts only a $k$-suffix of a blockchain and outputs the same value that $Q$ would have output if it had been evaluated on a chain with that suffix.

**Algorithm 3.** The Prove algorithm for the NIPoPoW protocol

```
1: function $\mathsf{Prove}_{m,k}(\mathcal{C})$
2:     $B \leftarrow \mathcal{C}[0]$                                        ▷ Genesis
3:     for $\mu = |\mathcal{C}[-k-1].\mathsf{interlink}|$ down to 0 do
4:         $\alpha \leftarrow \mathcal{C}[:-k]\{B:\}{\uparrow^{\mu}}$
5:         $\pi \leftarrow \pi \cup \alpha$
6:         if $m < |\alpha|$ then
7:             $B \leftarrow \alpha[-m]$
8:         end if
9:     end for
10:    $\chi \leftarrow \mathcal{C}[-k:]$
11:    return $\pi\chi$
12: end function
```

**The Concrete Prover.** The NIPoPoW prover construction is shown in Algorithm 3. The honest prover is supplied with an honestly adopted chain $\mathcal{C}$ and security parameters $m, k$ and returns proof $\pi\chi$, which is a chain. The suffix $\chi$ is the last $k$ blocks of $\mathcal{C}$. The prefix $\pi$ is constructed by selecting various blocks from $\mathcal{C}[:-k]$ and adding them to $\pi$, which consists of a number of blocks for every level $\mu$ from the highest level $|\mathcal{C}[-k].\mathsf{interlink}|$ down to 0. At the highest possible level at which at least $m$ blocks exist, all these blocks are included. Then, inductively, for every superchain of level $\mu$ that is included in the proof, the suffix of length $m$ is taken. Then the underlying superchain of level $\mu - 1$ spanning from this suffix until the end of the blockchain is also included. All the $\mu$-superblocks which are within this range of $m$ blocks will also be $(\mu - 1)$-superblocks and so we do not want to keep them in the proof twice (we use the union set notation to indicate this). Each underlying superchain will have $2m$ blocks in expectation and always at least $m$ blocks. This is repeated until level $\mu = 0$ is reached. Note that no check is necessary to make sure the top-most level has at least $m$ blocks, even though the verifier requires this. The reason is the following: Assume the blockchain has at least $m$ blocks in total. Then, when a superchain of level $\mu$ has less than $m$ blocks in total, these blocks will all be necessarily included into the proof by a lower-level superchain $\mu - i$ for some $i > 0$. Therefore, it does not hurt to add them to $\pi$ earlier.

Figure 3 contains an example proof constructed for parameters $m = k = 3$. The top superchain level which contains at least $m$ blocks is level $\mu = 2$. For the $m$-sized suffix of that level, 6 blocks of superblock level 1 are included to span the same range ($2m$ blocks at this level). For the last 3 blocks of the 1-superchain, blocks of level 0 spanning the same range are included (again $2m$ blocks at this level). Note that the superchain at a lower levels may reach closer to the end of the blockchain than a higher level. Level 3 was not used, as it does not yet have a sufficient number of blocks.

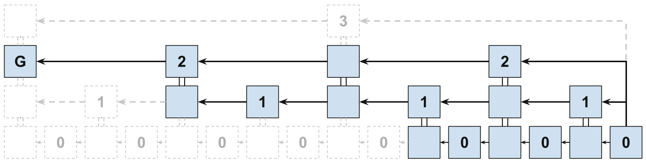

**Fig. 3.** NIPoPoW *prefix* $\pi$ for $m = 3$. It includes the Genesis block $G$, three 2-superblocks, six 1-superblocks, and six 0-blocks.

**Algorithm 4.** The algorithm implementation for the $\geq_m$ operator to compare two proofs in the NIPoPoW protocol parameterized with security parameter $m$. Returns *true* if the underlying chain of player $A$ is deemed longer than the underlying chain of player $B$.

```
1: function best-arg$_m(\pi, b)$
2:     $M \leftarrow \{\mu : |\pi\uparrow^{\mu} \{b:\}| \geq m\} \cup \{0\}$          ▷ Valid levels
3:     return $\max_{\mu \in M}\{2^{\mu} \cdot |\pi\uparrow^{\mu} \{b:\}|\}$        ▷ Score for level
4: end function
5: operator $\pi_A \geq_m \pi_B$
6:     $b \leftarrow (\pi_{\mathcal{A}} \cap \pi_B)[-1]$                           ▷ LCA
7:     return best-arg$_m(\pi_A, b) \geq$ best-arg$_m(\pi_B, b)$
8: end operator
```

**The Concrete Verifier.** The $\geq_m$ operator which performs the comparison of proofs is presented in Algorithm 4. It takes proofs $\pi_A$ and $\pi_B$ and returns *true* if the first proof is winning, or *false* if the second is winning. It first computes the LCA block $b$ between the proofs. As parties $A$ and $B$ agree that the blockchain is the same up to block $b$, arguments will then be taken for the diverging chains after $b$. An *argument* is a subchain of a proof provided by a prover such that its blocks are after the LCA block $b$ and they are all at the same level $\mu$. The best possible argument from each player's proof is extracted by calling the best-arg$_m$ function. To find the best argument of a proof $\pi$ given $b$, best-arg$_m$ collects all the indices $\mu$ which point to superblock levels that contain valid arguments after block $b$. Argument validity requires that there are at least $m$ $\mu$-superblocks following block $b$, which is captured by the comparison $|\pi\uparrow^{\mu} \{b:\}| \geq m$. 0 is always considered a valid level, regardless of how many blocks are present there. These level indices are collected into set $M$. For each of these levels, the score of their respective argument is evaluated by weighting the number of blocks by the level as $2^{\mu}|\pi\uparrow^{\mu} \{b:\}|$. The highest possible score across all levels is returned. Once the score of the best argument of both $A$ and $B$ is known, they are directly compared and the winner returned. An advantage is given to the first proof in case of a tie by making the $\geq_m$ operator favour the adversary $\mathcal{A}$.

Looking ahead, the core of the security argument will be that, given a block $b$, it will be difficult for a mining minority adversary to produce blocks descending from $b$ faster than the honest party. This holds for blocks of any level.

## 5 Analysis

We now give a sketch indicating why our construction is secure. The fully formal security proof, together with a detail in the construction which ensures statistical *goodness* and is necessary for withstanding full 1/2 adversaries, appears in the appendix.

**Theorem 1 (Security).** *Assuming honest majority, the Non-interactive Proofs of Proof-of-Work construction for computable $k$-stable monotonic suffix-sensitive predicates is secure with overwhelming probability in $\kappa$.*

*Proof (Sketch).* Suppose an adversary produces a proof $\pi_{\mathcal{A}}$ and an honest party produces a proof $\pi_B$ such that the two proofs cause the predicate $Q$ to evaluate to different values, while at the same time all honest parties have agreed that the correct value is the one obtained by $\pi_B$. Because of Bitcoin's security, $\mathcal{A}$ will be unable to make these claims for an actual underlying 0-level chain.

We now argue that the operator $\leq_m$ will signal in favour of the honest parties. Suppose $b$ is the LCA block between $\pi_{\mathcal{A}}$ and $\pi_B$. If the chain forks at $b$, there can be no more adversarial blocks after $b$ than honest blocks after $b$, provided there are at least $k$ honest blocks (due to the Common Prefix property). We will now argue that, further, there can be no more disjoint $\mu_{\mathcal{A}}$-level superblocks than honest $\mu_B$-level superblocks after $b$.

To see this, let $b$ be an honest block generated at some round $r_1$ and let the honest proof be generated at some round $r_3$. Then take the sequence of consecutive rounds $S = (r_1, \cdots, r_3)$. Because the verifier requires at least $m$ blocks from each of the provers, the adversary must have $m$ $\mu_{\mathcal{A}}$-superblocks in $\pi_{\mathcal{A}}\{b :\}$ which are not in $\pi_B\{b :\}$. Therefore, using a negative binomial tail bound argument, we see that $|S|$ must be long; intuitively, it takes a long time to produce a lot of blocks $|\pi_{\mathcal{A}}\{b :\}|$. Given that $|S|$ is long and that the honest parties have more mining power, they must have been able to produce a longer $\pi_B\{b :\}$ argument (of course, this comparison will have the superchain lengths weighted by $2^{\mu_{\mathcal{A}}}, 2^{\mu_B}$ respectively). To prove this, we use a binomial tail bound argument; intuitively, given a long time $|S|$, a lot of $\mu_B$-superblocks $|\pi_B\{b :\}|$ will have been honestly produced.

We therefore have a fixed value for the length of the adversarial argument, a negative binomial random variable for the number of rounds, and a binomial random variable for the length of the honest argument. By taking the expectations of the above random variables and applying a Chernoff bound, we see that the actual values will be close to their means with overwhelming probability, completing the proof. $\square$

We formalize the above proof sketch in the full version of this paper.

Lastly, the following theorem illustrates that our proofs are succinct. Intuitively, the number of levels exchanged is logarithmic in the length of the chain, and the number of blocks in each level is constant. The formal proofs are included in the Appendix.

**Theorem 2 (Optimistic succinctness).** *In an optimistic execution, Non-Interactive Proofs of Proof-of-Work produced by honest provers are succinct with the number of blocks bounded by* $4m\log(|\mathcal{C}|)$*, with overwhelming probability in* $m$*.*

## 6 Non-interactive Blockchain *infix* proofs

In the main body we have seen how to construct proofs for suffix predicates. As mentioned, the main purpose of that construction is to serve as a stepping stone for the construction of this section that presents a more useful class of proofs. This class of proofs allows proving more general predicates that can depend on multiple blocks even buried deep within the blockchain.

More specifically, the generalized prover for *infix proofs* allows proving any predicate $Q(\mathcal{C})$ that depends on a number of blocks that can appear anywhere within the chain (except the $k$ suffix for stability). These blocks constitute a *subset* $\mathcal{C}'$ of blocks, the *witness*, which may not necessarily form a chain. This allows proving useful statements such as, for example, whether a transaction is confirmed. We next formally define the class of predicates that will be of interest.

**Definition 6 (Infix sensitivity).** *A chain predicate* $Q_{d,k}$ *is* infix sensitive *if it can be written in the form*

$$Q_{d,k}(\mathcal{C}) = \begin{cases} \text{true}, \textit{ if } \exists \mathcal{C}' \subseteq \mathcal{C}[: -k] : |\mathcal{C}'| \leq d \wedge D(\mathcal{C}') \\ \text{false}, \textit{ otherwise} \end{cases}$$

*where* $D$ *is an arbitrary efficiently computable predicate such that, for any block sets* $\mathcal{C}_1 \subseteq \mathcal{C}_2$ *we have that* $D(\mathcal{C}_1) \rightarrow D(\mathcal{C}_2)$*.*

Note that $\mathcal{C}'$ is a blockset and may not necessarily be a blockchain. Furthermore, observe that for all blocksets $\mathcal{C}' \subseteq \mathcal{C}$ we have that $Q(\mathcal{C}') \rightarrow Q(\mathcal{C})$. This will allow us to later argue that adding more blocks to a blockchain cannot invalidate its witness.

Similarly to suffix-sensitive predicates, infix-sensitive predicates $Q$ can be evaluated very efficiently. Intuitively this is possible because of their localized nature and dependency on the $D(\cdot)$ predicate which requires only a small number of blocks to conclude whether the predicate should be true.

**Example.** We next show how to express the predicate that asks whether a certain transaction with id *txid* has been confirmed as an infix sensitive predicate. We define the predicate $D^{txid}$ that receives a single block and tests whether a transaction with id *txid* is included. The predicate $Q^{txid}_{1,k}$ is defined as in Definition 6 using the predicate $D^{txid}$ and the parameter $k$ which in this case determines the desired stability of the assertion that *txid* is included (e.g., $k = 6$). $Q$

alone proves that a particular block is included in the blockchain. Some auxiliary data is supplied by the prover to aid the provability of transaction inclusion: the Merkle Tree proof-of-inclusion path to the transactions Merkle Tree root, similar to an SPV proof. This data is logarithmic in the number of transactions in the block and, hence, constant with respect to blockchain size. In case of a vendor awaiting transaction confirmation to ship a product, the proof that a certain transaction paid into a designated address for the particular order is sufficient. In this scheme it is impossible to determine whether the money has subsequently been spent in a future block, and so must only be used by the vendor holding the respective secret keys.

In the above example, note that if the verifier outputs *false*, this behavior will generally be inconclusive in the sense that the verifier could be outputting *false* either because the payment has not yet been confirmed or because the payment was never made.

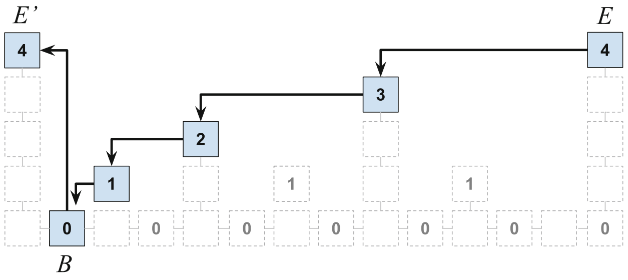

**Fig. 4.** An infix proof descend. Only blue blocks are included in the proof. Blue blocks of level 4 are part of $\pi$, while the blue blocks of level 1 through 3 are produced by followDown to get to the block of level 0 which is part of $\mathcal{C}'$. (Color figure online)

**Algorithm 5.** The Prove algorithm for infix proofs

```
1: function ProveInfix$_{m,k}$($\mathcal{C}$, $\mathcal{C}'$, height)
2:     $aux \leftarrow \emptyset$
3:     $(\pi, \chi) \leftarrow$ Prove$_{m,k}(\mathcal{C})$                ▷ Start with a suffix proof
4:     for $B \in \mathcal{C}'$ do
5:         for $E \in \pi$ do
6:             if height$[E] \geq$ height$[B]$ then
7:                 $aux \leftarrow aux \cup$ followDown$(E, B,$ height$)$
8:                 break
9:             end if
10:        end for
11:    end for
12:    return $(aux \cup \pi, \chi)$
13: end function
```

The construction of these proofs is shown in Algorithm 5. The infix prover accepts two parameters: The chain $\mathcal{C}$ which is the full blockchain and $\mathcal{C}'$ which is a sub-blockset of the blockchain and whose blocks are of interest for the predicate in question. The prover calls the previous suffix prover to produce a proof as usual. Then, having the prefix $\pi$ and suffix $\chi$ of the suffix proof in hand, the infix prover adds a few auxiliary blocks to the prefix $\pi$. The prover ensures that these auxiliary blocks form a chain with the rest of the proof $\pi$. Such auxiliary blocks are collected as follows: For every block $B$ of the subset $\mathcal{C}'$, the immediate previous ($E'$) and next ($E$) blocks in $\pi$ are found. Then, a chain of blocks $R$ which connects $E$ back to $B$ is found by the algorithm $\mathsf{followDown}$. If $E'$ is of level $\mu$, there can be no other $\mu$-superblock between $B$ and $E'$, otherwise it would have been included in $\pi$. Therefore, $B$ already contains a pointer to $E'$ in its interlink, completing the chain.

The way to connect a superblock to a previous lower-level block is implemented in Algorithm 6. Block $B'$ cannot be of higher or equal level than $E$, otherwise it would be equal to $E$ and the $\mathsf{followDown}$ algorithm would return. The algorithm proceeds as follows: Starting at block $E$, it tries to follow a pointer to as far as possible. If following the pointer surpasses $B$, then the procedure at this level is aborted and a lower level is tried, which will cause a smaller step within the skiplist. If a pointer was followed without surpassing $B$, the operation continues from the new block, until eventually $B$ is reached, which concludes the algorithm.

**Algorithm 6.** The followDown function which produces the necessary blocks to connect a superblock $E$ to a preceeding regular block $B$.

```
1: function followDown($E$, $B$, height)
2:     $aux \leftarrow \emptyset$; $\mu \leftarrow level(E)$
3:     while $E \neq B$ do
4:         $B' \leftarrow \mathsf{blockById}[E.\text{interlink}[\mu]]$
5:         if $\mathsf{height}[B'] < \mathsf{height}[B]$ then
6:             $\mu \leftarrow \mu - 1$
7:         else
8:             $aux \leftarrow aux \cup \{E\}$
9:             $E \leftarrow B'$
10:        end if
11:    end while
12:    return $aux$
13: end function
```

An example of the output of $\mathsf{followDown}$ is shown in Fig. 4. This is a portion of the proof shown at the point where the superblock levels are at level 4. A descend to level 0 was necessary so that a regular block would be included in the chain. The level 0 block can jump immediately back up to level 4 because it has a high-level pointer.

The verification algorithm must then be modified as in Algorithm 7.

The algorithm works by calling the suffix verifier. It also maintains a blockDAG collecting blocks from all proofs (it is a DAG because *interlink* can be adversarially defined in adversarially mined blocks). This DAG is maintained in the blockById hashmap. Using it, ancestors uses simple graph search to extract the set of ancestor blocks of a block. In the final predicate evaluation, the set of ancestors of the best blockchain tip is passed to the predicate. The ancestors are included to avoid an adversary who presents an honest chain but skips the blocks of interest. In particular, such an adversary would work by including a complete suffix proof, but "forgetting" to include the blocks generated by followDown for the infix proof pertaining to blocks in $\mathcal{C}'$.

**Algorithm 7.** The verify algorithm for the NIPoPoW infix protocol

```
1: function ancestors($B$, blockById)
2:     if $B$ = Gen then
3:         return $\{B\}$
4:     end if
5:     $\mathcal{C} \leftarrow \emptyset$
6:     for id $\in B$.interlink do
7:         if id $\in$ blockById then
8:             $B' \leftarrow$ blockById[id]
9:             $\mathcal{C} \leftarrow \mathcal{C} \cup$ ancestors($B'$, blockById)        ▷ Collect into DAG
10:        end if
11:    end for
12:    return $\mathcal{C} \cup \{B\}$
13: end function
14: function verify-infx$^D_{\ell,m,k}(\mathcal{P})$
15:    blockById $\leftarrow \emptyset$
16:    for $(\pi, \chi) \in \mathcal{P}$ do
17:        for $B \in \pi$ do
18:            blockById[id($B$)] $\leftarrow B$
19:        end for
20:    end for
21:    $\tilde{\pi} \leftarrow$ best $\pi \in \mathcal{P}$ according to suffix verifier
22:    return $D$(ancestors($\tilde{\pi}[-1]$, blockById))
23: end function
```

**Acknowledgements.** The authors wish to thank Giorgos Panagiotakos at the University of Edinburgh, Peter Gaži at IOHK, and Nikos Leonardos at the University of Athens for their insights regarding the attack against the interactive Proofs of Proof-of-Work scheme. We also wish to thank Giorgos Christoglou at Imperial College London and Kostis Karantias at the University of Ioannina for their insights and comments while they implemented the schemes described in this paper as part of their master theses, as well as Sebastien Guillemot at Emurgo R&D who provided valuable feedback. All three of them read many versions of this paper in detail and their contributions helped significantly shape and improve it.

## References

1. Apostolaki, M., Zohar, A., Vanbever, L.: Hijacking bitcoin: routing attacks on cryptocurrencies. In: 2017 IEEE Symposium on Security and Privacy (SP), pp. 375–392. IEEE (2017)
2. Back, A., et al.: Enabling blockchain innovations with pegged sidechains (2014). http://www.opensciencereview.com/papers/123/enablingblockchain-innovations-with-pegged-sidechains
3. Bellare, M., Rogaway, P.: Random oracles are practical: a paradigm for designing efficient protocols. In Proceedings of the 1st ACM Conference on Computer and communications security, pp. 62–73. ACM (1993)
4. Bonneau, J., Miller, A., Clark, J., Narayanan, A., Kroll, J.A., Felten, E.W.: Sok: research perspectives and challenges for bitcoin and cryptocurrencies. In: 2015 IEEE Symposium on Security and Privacy (SP), pp. 104–121. IEEE (2015)
5. Buterin, V., et al.: A next-generation smart contract and decentralized application platform. white paper (2014)
6. Douceur, J.R.: The sybil attack. In: Druschel, P., Kaashoek, F., Rowstron, A. (eds.) IPTPS 2002. LNCS, vol. 2429, pp. 251–260. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45748-8_24
7. Garay, J., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: analysis and applications. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9057, pp. 281–310. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46803-6_10
8. Gazi, P., Kiayias, A., Zindros, D.: Proof-of-stake sidechains. In: IEEE Symposium on Security & Privacy (2019)
9. Heilman, E., Kendler, A., Zohar, A., Goldberg, S.: Eclipse attacks on bitcoin's peer-to-peer network. In: USENIX Security Symposium, pp. 129–144 (2015)
10. Maurice Herlihy. Atomic cross-chain swaps. arXiv preprint arXiv:1801.09515 (2018)
11. Kiayias, A., Lamprou, N., Stouka, A.-P.: Proofs of proofs of work with sublinear complexity. In: Clark, J., Meiklejohn, S., Ryan, P.Y.A., Wallach, D., Brenner, M., Rohloff, K. (eds.) FC 2016. LNCS, vol. 9604, pp. 61–78. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53357-4_5
12. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Pomerance, C. (ed.) CRYPTO 1987. LNCS, vol. 293, pp. 369–378. Springer, Heidelberg (1988). https://doi.org/10.1007/3-540-48184-2_32
13. Miller, A.: The high-value-hash highway, bitcoin forum post (2012)
14. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
15. Nolan, T.: Alt chains and atomic transfers, May 2013. bitcointalk.org
16. Pass, R., Seeman, L., Shelat, A.: Analysis of the blockchain protocol in asynchronous networks. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10211, pp. 643–673. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56614-6_22
17. Pass, R., Shi, E.: Fruitchains: a fair blockchain. In: Proceedings of the ACM Symposium on Principles of Distributed Computing, pp. 315–324. ACM (2017)
18. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Commun. ACM **33**(6), 668–676 (1990)
19. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper **151**, 1–32 (2014)